5-2013

Real Time Algorithmic Musical Variation Using Style

Johnathan Pagnutti
University of New Orleans

Real Time Algorithmic Musical Variation Using Style

An Honors Thesis

Presented to

the Department of Computer Science

of the University of New Orleans

In Partial Fulfillment

of the Requirements for the Degree of

Bachelor of Science, with University Honors

and Honors in Computer Science

by

Johnathan Pagnutti

May 2013

Table of Contents

List of Tables

List of Figures

Abstract

My honors thesis aims to develop a program that could assist with music composition, or even write interesting music on its own. The starting point is a melody and chord progression composed by a musician, which is the same amount of information that a Jazz performer might get with a lead sheet. Then the computer is tasked with writing a harmony to the melody. The harmony is not only based on the melody and chords, but also on what the program user might want to hear. The program can be provided, in real time, with some descriptors of what a user might want to hear next- such as the words “bright” and “happy”, and the resultant music will be a little upbeat and in a major key.

The program accomplishes this through use of a hidden Markov model, which is a collection of probabilities for selecting which composition rules to use. Composition rules will affect how the harmony is voiced (from block chords to syncopation to arpeggios), the tempo of the theme, the key of the theme, or some other quality of the music. Each word the user inputs changes the probabilities that the next rules will use.

Obvious applications for this program are primarily in the entertainment industry. Video games and other electronic entertainment are prime candidates for this system, allowing for unique and fitting music to be generated based on some of the events occurring in the game.

Introduction

Music is often referred to as an enigma of sorts. This is strange, as nearly every single person on the planet has heard a song at least once or twice. Yet, when we are pressed to describe music, we end up using abstract terms, such as ‘upbeat’, ‘bright’, ‘dark’, ‘tranquil’, ‘lush’, ‘thin’, and so forth. This sort of ill-defined and abstract grammar, that forms the backbone of how humans talk about music, seems to fly in the face of a precise, clean definition that might be understood and acted upon computationally.

However, I propose that a computer can react appropriately to such stylistic, high-level descriptions of music, and use that information to compose a new variation on a pre-composed musical theme that reflects the quality given by the input, in real time. This is accomplished by augmenting computational models with concepts found in Western tonal music theory.

Composition of variations on a theme is likened to an interweaving state system, with each variation of a theme as a state. By enforcing a derivative of the Markov property on this system, and modifying the probability to change to a particular next state based on the input, then the resultant system, as it changes state, reflects the qualities listed as input. The system is computationally simple enough to allow for state changes to happen in real time. The system plays each new variation as it is composed by sending MIDI data to the sound card.

This paper outlines such a system, taking into consideration how the data is structured, how the process of writing a new theme can be broken into small parts that can each be performed in parallel, how the theme can be changed from variation to

variation and how the hidden Markov model is structured. First, however, it is necessary to provide a brief overview of both probability and Markov models, as well as elements of Western tonal harmony. The overview is by no means comprehensive, it is designed to allow readers from the disciplines of music and of computer science, as well as anyone else, to follow along. In the next section, the core algorithm and process is discussed. In section four, the design of the software system that implements the core algorithm is explained. Section five talks about how some concepts from Western tonal music are employed, then provides a quick analysis of some of the output of the program, showing how the input is reflected in the output. After that, the process is viewed from a computer science and information theory angle, giving an analysis of the transition tables used in the core algorithm. Finally, implications for further research are noted, as well as applications for the software system.

Background and Motivation

Algorithmic and computational composition of music has a longer history than what might be expected. In 1955, Hiller and Isaacson began investigating techniques to generate music using the ILLIAC, a computer on the campus of the University of Illinois. The first objective of the experiments was to “demonstrate that technical musical concepts could be translated into computer language to produce musical output” [13]. Some basic counter point rules were used to generate a cantus firmi, and then four-part first-species counterpoint. Robert Gjerdingen would later argue that later work based on the static rules-based system to be unmusical, since it treated counterpoint as a problem to be solved, rather than an “aesthetic utterance” [14]. However, unfortunately

for Robert, the use of rules to algorithmically compose pieces goes back even further than the ILLIAC.

As David Cope notes in Computer Models of Musical Creativity [7], algorithmic composition actually predates computer algorithms. The Musikalisches Würfelspiel, German for “musical dice game”, was a popular dice game in the 18th century in Western Europe. The concept was simple- a composer would devise a set of small musical ideas. Then, after a random number was picked, based on what was last played, the composer would then play a new segment. The way these games worked is similar to sentence construction, for example, a composer (of sentences) may write down:

```
1. The horse trotted though green grass.
2. The pig ran through brown mud.
3. A chicken picked at yellow grain.
```

A three sided die could be rolled 6 times, generating the numbers 1, 3, 2, 1, 3, 1. This would yield a new sentence: “The chicken ran through yellow grass.” The resultant sentence might not be the most elegant, but it does communicate a new idea that was a combination of older ones. I took a lot of inspiration from this idea, as have many other researchers in algorithmically composed music.

As the Musikalisches Würfelspiel hints at, minor structural changes to the way a piece of music is composed can drastically change the output. Bruce Jacob articulates this property of music [10], showcasing how, when performing just a few minor transformations to a theme, he could achieve drastically different output.

The other property the Musikalisches Würfelspiel seems to describe is the Markov property—that the next musical phrase is based on the precieding musical

phrase. However, the use of Markov structures to try and control high level properties of music has, so far, been neglected. Markov chains have been used to control low level properties, such as pitch or rhythym while seeding high level properties over to other algorithmic techniques, such as genetic algorithms [15].

There is a rational for this, as George Papadopoulos and Geraint Wiggins note that Markov chains, on their own, have trouble varying away from a rigid structure to try and account for a new motif or idea [16]. However, if the chains themselves were to get modified by an outside source, and modified in such a way that they could lean towards a new style, than this consideration might be overcome.

So, I attempted to use Markov chains to manipulate higher level musical concepts and also permute the Markov chains based on user input to allow for a larger range of expression than is usually attributed to such a system. Markov chains, and other mathmatical models of music composition, are fast to compute, so by allowing such a system to compose in real time, I could also create a utility that could be useful in providing music alongside dynamic content or as a composition aid.

Language, Environment and Additional Libraries

The language used to implement the software system is the ANSI (American National Standards Institute) C programing language, as described by the International Standard for Organization in 2005 [1]. However, C, in its native state, does not provide tools for parallelism and multiprocessing [2], which help the software system run in real time, nor does it provide a facility to communicate with the sound drivers on a computer. With this in mind, the Windows 7 64-bit operating system was employed to run the system. The windows.h header file and associated library, Kernel32, provides an

intuitive set of functions for parallel processing and multithreading. This external file and library are part of the Windows Operating System.

In addition, the PortMidi library is used. PortMidi is part of PortMedia, a suite of open-source libraries and API's [3], for various audio-based applications. The library in question, PortMidi, is designed for real-time MIDI input and output. PortMidi is used by the program to 'play' the variations it has composed.

II: Overview of Markov Chains and Music Theory

A Markov chain is nothing more than a network of states. Each state may be able to change to a different state, usually at some time. When this state change is only based on the current state, the system of states is called a 1-dimensional Markov chain [4]. You can express the likelihood of any individual state moving to another state as a probability in terms of a fraction. The fraction is just the number of ways one state cam move to another state over the total number of ways that state could change. For example, if a state *n* could change to one of three other states, *x*, *y*, *z*, then *n* would have a 1/3 chance of moving to either *x, y*, or *z*. If the state change to *x* was weighted, such that *n* has what amounts to two ways to change state to *x,* and only one way to change state to *y* or *z*, then *n* would have a 1/2 probability of moving to *x*, (2 out of 4 ways to change state go to *x*) and a 1/4 probability to move to *y* or *z*.

Change of state, therefore, can be viewed as a probability. It then bears to reason that if two separate Markov Chains both change state at some time *t*, the likelihood of both chains moving to a certain state *n* could be expressed as the odds of two independent events occurring. As such, if P(*x*) is a function that gives the

probability of *x* occurring, then the probability of A and B occurring is P(A)P(B), if A and B are independent.

These two concepts, independence in probability and Markov Chains, are all the mathematics required to understand the process and algorithm employed. The discussion of music theory, however, is going to be slightly more involved. It is important to note here that the concepts discussed are trends; for any example given, there is a composer who wrote a counter example.

Music is nothing more than organized sound [6]. We organize sound according to four broad categories- pitch, rhythm, timbre and loudness. Out of these four qualities, only three will be discussed. Timbre falls outside of the range of the program in its current state, and will be reviewed later on.

Loudness is the simplest. In music, we categorize various levels of volume into six general categories: pianissimo (softest), piano (soft), mezzo-piano (medium-soft), mezzo-forte (medium-loud), forte (loud), and fortissimo (loudest). An increase in volume is called a crescendo, a decrease in volume a decrescendo.

Rhythm is slightly more complicated. At its most basic, rhythm is nothing more than organizing when sounds are played through time. A measure is a segment of time, broken into a number of beats. The amount of time each beat occupies is set by the tempo, which is nothing more than the number of beats per second. Beats are organized in a measure either in duple patterns (based on 2) or triple patterns (based on 3).

Both types of patterns have an intrinsic accent pattern, or levels of importance our ears assign to certain beats in a measure. The beats that are considered important

are strong beats, the less important beats are called weak beats. For duple pattern measures, every four subdivisions is considered strong, provided there are four beats in a measure. If there are only two beats, every two subdivisions is considered strong. For triple pattern measures, every three. Finally, notes that occur in between beats (off-beat notes) add excitement and forward motion to a musical idea.

The smallest beat length the software handles is a 32nd note, which would be 1/32 of a measure. All greater beat lengths, such as a half note (1/2 a measure) or an eighth note (1/8 of a measure) can be considered as being composed of 32nd notes, with more complex rhythms falling outside the scope of this program.

We organize pitch into 12 tones: A, A#/Bb, B, C, C#/Db, D, D#/Eb, E, F, F#/Gb, G, G#/Ab. These tones repeat, with the first repetition said to be an octave higher than the last. We form keys, or collections of related tones based on scales. Scales are a series of these 12 tones that follow a particular spacing relationship (the space between two notes is called an interval). There are an almost infinite number of possible keys, but the program only deals with two-- the major key and the minor key, so the major scale and minor scale, respectively.

For a major scale, the intervals used are a major 2nd (two tones away) and a minor second (one tone away). Starting from a selected pitch, the tones used are a major second, major second, minor second, major second, and minor second. For a minor scale, starting from the selected pitch, the tones used are a major second, minor second, major second, major second, minor second, minor third (3 notes away) and a minor second. A detailed explanation of why falls outside the scope here, however, [6] is an excellent resource for further reading of tonal harmony.

Algorithm and Process

As brought up in the introduction, and mentioned in section II, the software system relies on Markov Chains to compose music. At its most basic, each quality of music that we want to vary in the output is mapped to a Markov Chain through the use of a transition matrix. The Markov Chain for the particular quality has *n* states, with each state being a different way to express that quality. The transition matrix, T, is an *n* x *n* square matrix such that each element of the transition matrix T_{ij}, such that *i* and *j* are integers between 0 and *n*, is the probability of state *j* occurring after state *i*. Therefore, each row of the matrix can be considered the state changes that state *i* can undergo. As such, all the elements of T_i must sum up to 1, expressed as:

$$T : T_{ij} = p_{ij} : \sum_{n} T_{ij} = 1$$

This leads to the use of several 1-dimensional Markov Chains to control how music is expressed, with more Markov chains allowing for greater control and level of expression to be considered. Therefore, for each high level style description, a set of Markov Chains can be created that model how each chosen musical quality trends through time for that particular style. Each style is expressed as a set S. Each element of S is a discrete random variable Y, such that the image of Y is the set of random variables, X_1, X_2, X_3,... that hold to the Markov Property described above, or concisely:

$$S := \{Y_1, Y_2, \dots, Y_n | Im(Y_n) = \Pr(X_{m+1} = x \mid X_m = x_m)\}$$

$$p_{ij} = \Pr(X_{n+1} = j \mid X_n = i)$$

$$S = \{T^1, T^2, \dots, T^k | T_{ij}^k = p_{ij}^k\}$$

This allows us to capture how an individual style descriptor effects musical output, but, this approach does not allow us to consider multiple style descriptors, nor

does it allow us to try and capture a unique sense of style that may vary from composer to composer. To compensate for multiple style descriptors being supplied for a new variation, say, both ‘dark’ and ‘serene’, we consider the joint probability of both sets of Markov Chains. For each set of style descriptors provided as input, a new set is created S’. The elements of S’ are the image of a random variable Y’, such that the values of Y’ obey a new transition matrix T’. T’ is similar to our previous matrix, T, in that it has the same number of elements, and each element is a probability of state *j* following state *i*, however each probability in T’ is the probability of T^{dark}_{ij} and T^{serene}_{ij} occurring, or $(p^{dark}_{ij})\ p^{serene}_{ij}$. This can be expressed as:

$$S' \coloneqq \{T^{input}\}$$

$$T' = \prod_{l} S'_l : T'_{ij} = \prod_{l} p^{l}_{ij}$$

This is different than standard matrix multiplication. It should be noted that this element by element style of multiplication only works for square matrices of the same size. This is not a problem to the software system at hand.

At this point, the program still follows the Markov Property, that the change of each state is only based on the current state and no information in the past. However, relaxing this constraint leads to interesting and desirable behavior. If it is proposed that the set W is stateful, that is, it stores past states, we can allot for more subtle and gradual changes in output when a new style is selected. Therefore, rather than creating a new matrix T’ when a new set of input is entered, it can be beneficial to instead consider T’ as product of every past state the system has undergone, expressed as:

$$T'_s = T'_{s-1} \prod_{l} S'_l$$

This provides the underlying framework of how each new variation is expressed in terms of the style parameters. However, there is no requirement that each Markov Chain will transfer to the same state at the same time, which will lead to a conflict of states. Some properties of musical expression may lend themselves to an average among all the chosen states (such as loudness), but others are more mutually exclusive (such as key). To make up for this, instead of storing transition tables for each style descriptor, it makes sense to instead describe them in terms of weight. Each element of some style parameter table T is, instead of p_{ij}, the number of ways state i could go to state j, allowing for floating point values-- so i may have 0.5 ways to follow j. Now, we calculate weight first, then calculate the normalized probabilities (probabilities out of one) for each element. This enforces that all table rows will sum up to 1, as per Figure 1, and also removes conflicting states, instead focusing on the ratios between similar state motion. So, finally, the set T' is expressed as:

$$\oplus_a A \coloneqq \{\frac{A_{a1}}{\sum_b A_{ab}}, \frac{A_{a2}}{\sum_b A_{ab}}, \dots, \frac{A_{ab}}{\sum_b A_{ab}}\}$$

$$S'' \coloneqq \{S_{weight}^{input}\}$$

$$W_s \coloneqq W_{s-1} \prod_l S''_l$$

$$T''_s = \oplus_i W_s$$

With this in mind, the core composition algorithm becomes rather simple. The resultant behavior may be complex, but only minor calculations are performed at each step. It also lends itself to being calculated in parallel, which is what allows a large amount of expressive parameters in music to be described in real time. The process, closer to how it is implemented in a computer is as follows, psudocode:

```
For each input_param:
     selection[][];
     weight[][];
     For each input_param.musical_quality:
          while i < input_param.musical_quality[i].length{
               while j < input_param.musical_quality[i].length{
                    weight[i][j] *=
                    input_param.musical_quality[i][j];
                    j++;
               }
               i++;
          }
     }
     while i < weight.length{
          total = 0, sum = 0;
          while j < weight[i].length{
               total += weight[i][j];
          }
          while j < weight[i].length{
               sum += weight[i][j];
               selection[i][j] = sum / total;
          }
     }
}
For each input_param:
     picked = rand();
     while selection[last_picked][i] < picked{
          i++;
     }
     last_picked = selection[last_picked][i];
     apply_to_theme(last_picked);
}
```

There are several key differences here than the equations given above. The most prominent is that each probability is expressed as a range between 0 and 1 as $[x_n , x_{n+1})$. This aids the selection process. Also, the order in which things are calculated is not entirely apparent from the equations given. However, this order does lend itself to threading, as will be later shown.

IV: Considerations from Music Theory

If we consider how a musical piece is expressed as a composite the number of individual qualities expressed, then the way the theme changes from variation to variation is a direct consequence of which qualities are changed from variation to variation. Therefore, it is important to pick qualities that reflect how you would want the theme to be varied. This leads to another interesting fork in philosophy, much like the decision to relax the Markov property before, there are an almost innumerable amount of different qualities a musical theme can have. To use smaller-level phenomena, such as the attack on each note, leads to new variations that are more subtly different. Larger scale phenomena, such as key, leads to more dramatic changes in content from variation to variation [6].

As such, I chose to use several examples of larger scale qualities to vary from theme to theme. The four I chose to use reflect three of the four main qualities of music: rhythm, pitch and loudness. The four qualities that the program changes from variation to variation are: Harmonic Voicing, Key, Tempo and Dynamic Level.

Harmonic Voicing is a combination of elements from both pitch and rhythm. This parameter takes the most inspiration from the Musikalisches Würfelspiel. Harmonic Voicing is simply the way a set of chords that go with a theme are voiced, from a simple idea of the root of each chord to complex multi-voice lines. For purposes of the program, however, Harmonic Voicing is always only going to be a single line. The way harmonies are voiced draws on several simple effects that are discussed in theoretical analysis-- for example, faster note motion generally generates excitement. Syncopated rhythms also tend to generate excitement and interest, pulling the piece along. Large

intervals between the notes voiced tends to generate a sense of grandness, and sparse use of notes implies a thinner, lighter sound. These are just a few examples, but by keeping in mind these sorts of ‘effects’ the states for the Harmonic Voicing table can be mapped back to trends implied by the input.

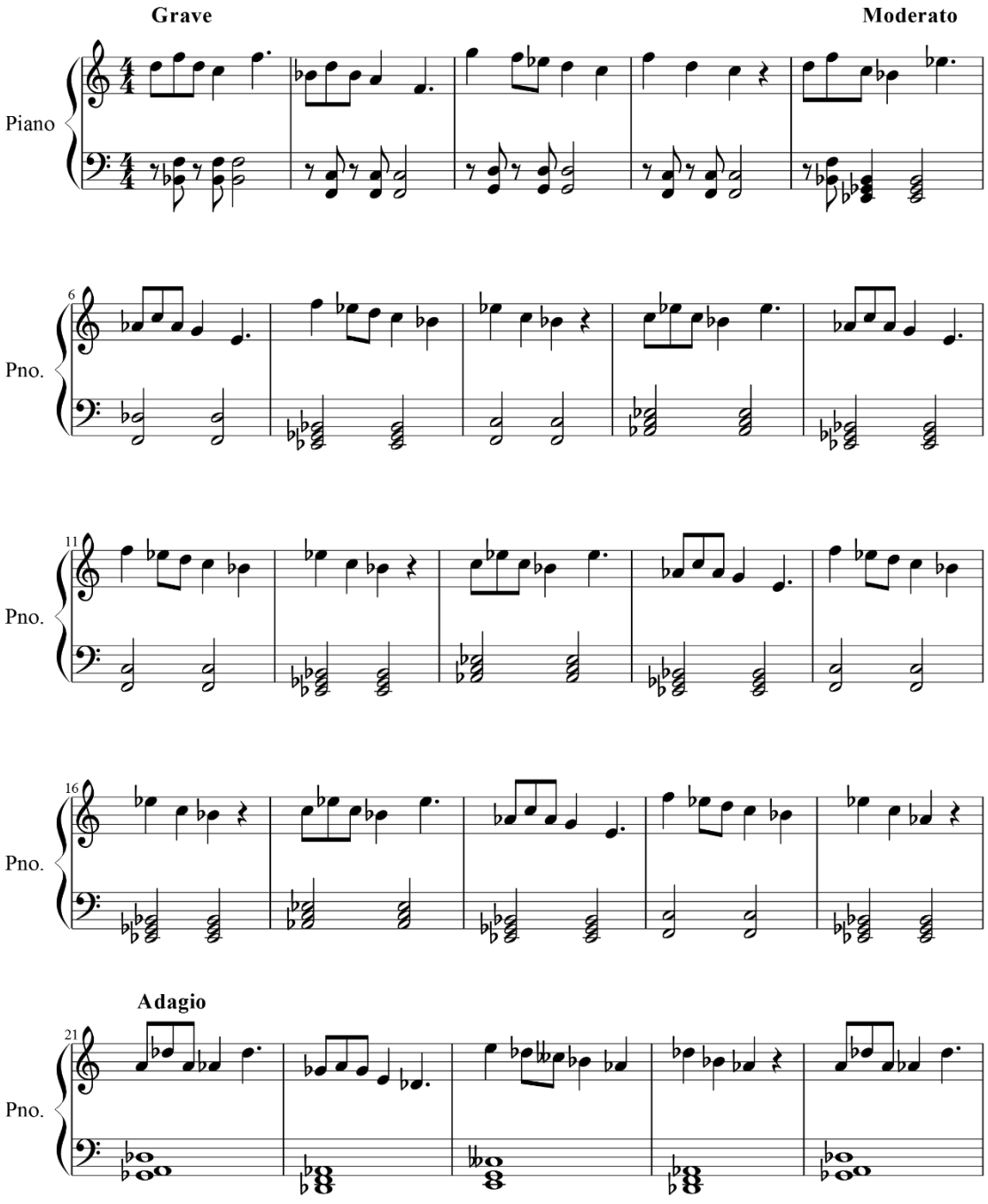
Key is the pitch collection that will be used for this variation of the theme. Although, there is a lot of overlap in the pitches chosen (there are 12 pitches and far more keys). Key is based in the intervals between the pitches, starting with the root pitch (the letter the key is named under). The intervals chosen follow a set pattern, and it is this pattern that differentiates between major keys and minor ones. For the software system, all 12 pitches were considered acceptable roots, with major and minor keys as used keys. Major keys tend to sound brighter than minor keys, with minor keys tending to sound darker than their major equivalents. In addition, each root of the scale is not created equal. Keys based in C tend to sound the brightest, with each letter getting darker until Db/C# which sounds the most dark [6]. The process of changing key is called transposition.

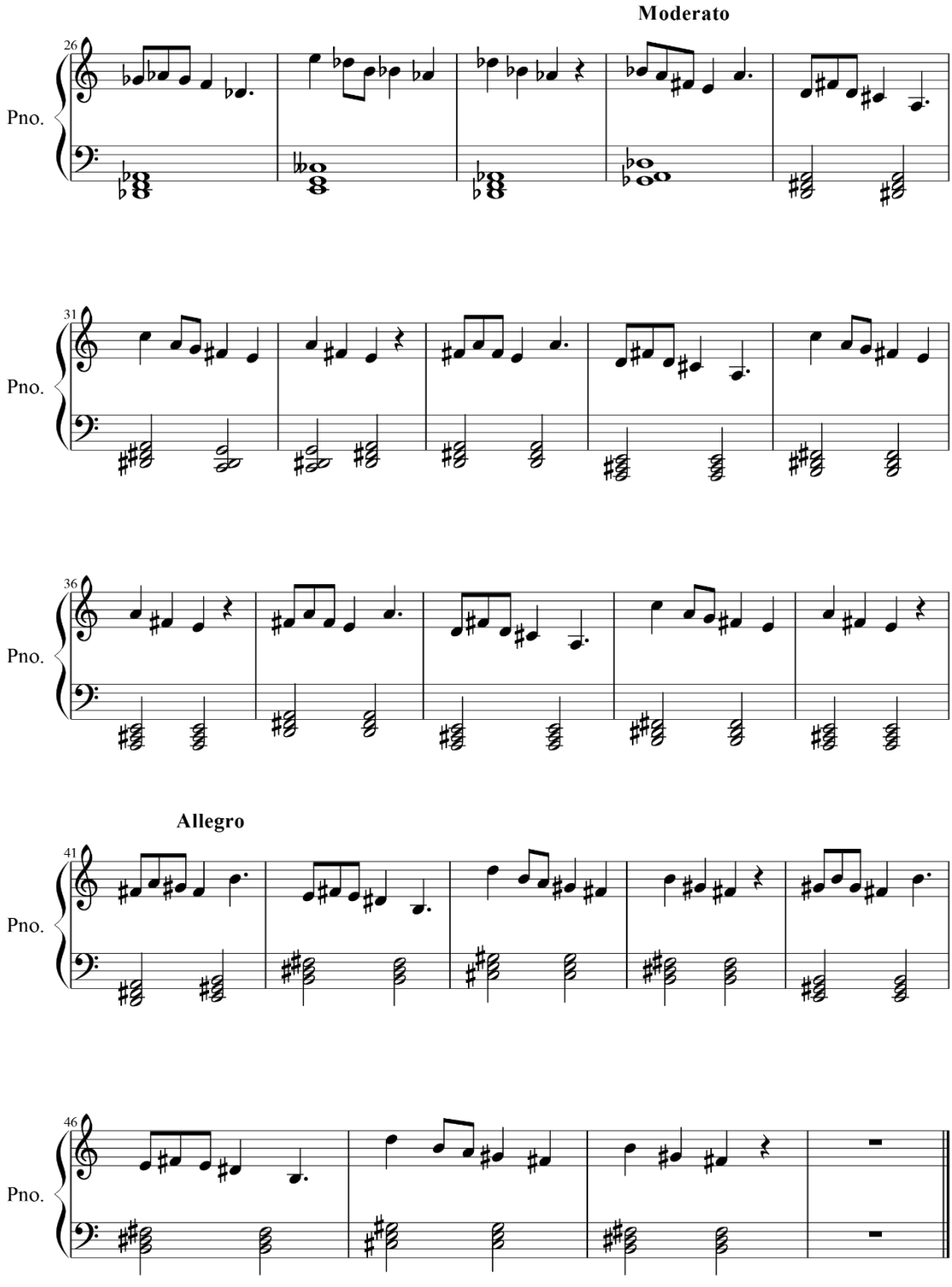
Tempo is, very simply, the speed of the beat. Faster tempos tend to produce more active and energetic music, whereas slower tempos tend towards calmer variations. Tempo is often notated not as a number in beats per minute, but as a descriptive phrase. As such, although there are tempos that are more common than others, but these are not the tempos that the software system considers. The tempos chosen were based on the tempos used in MuseScore, a free musical notation and composition tool [8].

Loudness, or dynamics, is exactly what one might expect- the quality of how loud the sound is. Loudness, in practice, has several layers that also relate to timbre: in a piece with multiple instruments, as some instruments get softer while others get louder, the net result is a change in timbre through a change in dynamics. The program only considers discrete dynamics- each new variation is at a set dynamic level, which is similar to the terraced dynamics of the baroque period of classical music [9]. The louder the variation, the more exciting and bombastic it sounds. The quieter a variation, the calmer and more serene it sounds. The software system only considers dynamics from pianissimo to fortissimo, with no gradual dynamic shifts.

It seems apparent that the software system is not actually changing much about the next variation. This is true-- the amount of change in each new variation is actually pretty slight. However, it is important to note that, as Bruce Jacob [10] found in his observations that slight change could lead to very musical different sounding output. Based on his observations, I also kept the changes between variations slight, which keeps the amount of calculations required for a new variation down, which allows for a new variation to be composed in real time.

To Brighten




Moderato
26
Pno.
31
Pno.
36
Pno.
Allegro
41
Pno.
46
Pno.

In these two pages, I converted some output from the program into sheet music. The title was more a play on the input I gave the program: for the first variation, I used the word “bright”. The second was created using “bright, serene”. The third and fourth were also created using “bright”.

The easiest way to tell variation changes is with the tempo changes, as each new variation has a new tempo. The theme was a simple 4 bar construct that follows a set I-V-iv-V chord pattern. In this example, there appears to be little change from theme to theme, as the way the harmony was chosen to be voiced turned out to be mostly the same.

However, the other qualities lead to a gradual “brightening” of the theme. To start, the theme is in a major key with an accompanying offbeat open fifth pattern. The tempo is, very slow (30 bpm). The key chosen is the key of Bb major, a very bright key.

In the first variation, the tempo speeds up substantially (108 bpm). The key chosen is, in fact, a darker key (the key of Ab major), but Ab and Bb are nearby keys (a major 2nd apart). I attribute this selection to the fact that it is hard to get brighter from Bb—only B is brighter. The key also remained major.

The harmonic voicing side bucked against the trend—for “bright” the trend was to try and select harmonic voicings with more activity. However, due to the tempo increase and the still fairly bright key, the resultant effect was a slightly brighter theme.

The next theme is the most interesting—perhaps due to the more complex input. The tempo decreases somewhat (54 bpm), but there is now a strong key shift to Gb minor. Gb and Bb share a third relation, so the key is still related. Harmonic voicing has changed somewhat, giving us whole notes. This changes the color of the piece by

a larger margin than what has so far been heard. Now that a minor key is being used, the new progression is i – V – IV – V, which, interestingly, contains more major chords than the prior theme.

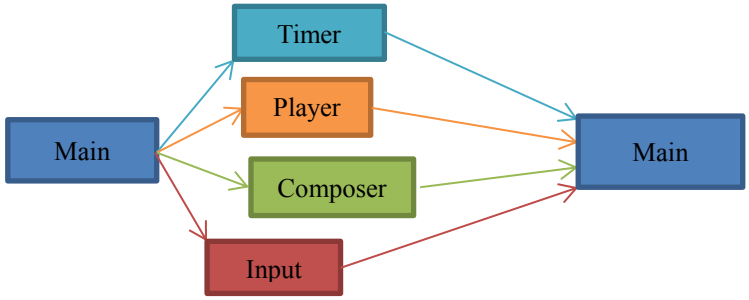
The next two changes show a brightening of this current state, shifting the key into D major, then E major, along with a tempo increase (108 bpm to 120 bpm). The conclusion, therefore, is actually very bright, due to the major tonality and fast tempo.

What is interesting to note is what isn't algorithmically coded. The "transition sections", where the theme doesn't seem to line up with any chords, are not programed. They are a pseudo-random result—I allowed the composer thread to directly modify the tracks while the tracks were still playing, with no use of locks or copies. The result is the strange transition from theme to theme, sometimes throwing off an entire theme repetition while the software system realigns itself.

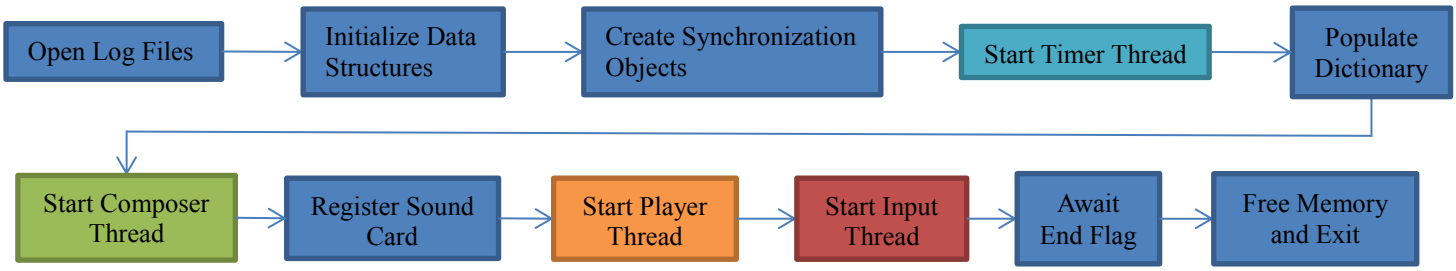
V: Software System

The question of real time music composition, when it comes to software, is actually a two sided problem. To truly show real-time output, the system must be able to both compose and play a segment of music more or less simultaneously, in addition to reacting to user input. As such, the software system can be organized into three main threads of execution: the composer, the player and input. Both the composer and the player's tasks can be broken down again into independent sections that can be performed simultaneously, thus adding sub-threads to each main thread. In addition, other common programmatic tasks, such as logging, can be threaded to relieve strain on the working threads. The first thread to start execution is main, and from there, all other programmatic tasks are started and performed, which can be categorized into

three groups: composer handling, player handling and input handling. Below is a flow chart representation of how the program executes. For each section (color), a more in-depth chart is provided along with a description. Each new element in the chart is a new programmatic task, with branches in the chart representing tasks that are performed in parallel.



Main



The main thread is the one that starts all other threads, main simply loads all the information from the data files into memory, allocates memory for the various data structures and initializes the state of the program. Main can be considered a “master” thread, as all other threads eventually spawn from it. Main is important, as it works as a sort of system fail-safe. After doing the work required for the program to run and starting threads, main watches a system failure flag. If this flag is thrown, the software system has hit a critical bug, and main attempts to gracefully close the system without

crashing. The work main performs is linear, there is no reason to use extra threads at this point, as initialization does not need to happen on demand. [22].

Main initializes the data structures used in the program. There are two core data structures, the `MusicalTheme` structure and the `style_hashtab` dictionary. `MusicalTheme` is the simpler of the two—the musical theme consists of several arrays and integers, breaking music into the tracks to play and various metadata associated with those tracks along with metadata that is important to all tracks. There are two tracks that are used: a melodic track, which plays the human composed melody and a harmonic track, which plays a pattern according to a set of human chosen chords and the current composition rules. The harmonic metadata arrays keep track of which chords to use for the patterns to play the harmonic voicing track. Global data for all tracks include the current key of the variation and the current dynamic level of the variation.

The `style_hashtab` dictionary is a hash table of each string that denotes a style. Styles are collections of weight tables that modify the state of the theme before rule selection. Each element in the `style_hashtab` points to a `style_param_entry`, which contains both the name of the style (used for user input) and a `StyleParameter` structure. Style parameters are a set of two dimensional double arrays that contain weights for each musical quality to vary.

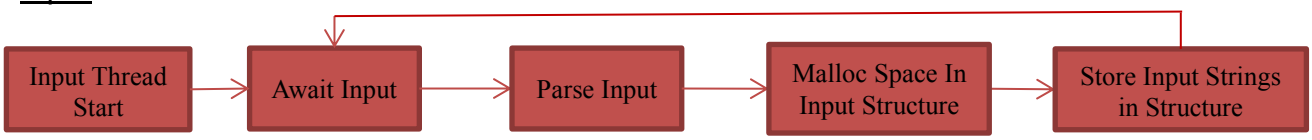
```
#bright#
:tempo:
0.25, 1.50, 1.00, 0.25, 0.25, 0.25, 0.25
0.25, 0.25, 1.50, 1.00, 0.25, 0.25, 0.25
0.25, 0.25, 0.25, 1.50, 1.00, 0.25, 0.25
0.25, 0.25, 0.25, 0.25, 1.50, 1.00, 0.25
0.25, 0.25, 0.25, 0.25, 0.25, 1.50, 1.00
0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 1.50
0.25, 0.25, 0.25, 0.25, 0.25, 1.25, 1.50
```

The dictionary is populated by main from a data file. The data file is a structured text file which lists the elements of the style parameter arrays along with the quality they map to and the name of the style. For example, Table 1, lists the table controlling tempo under the style name bright. The various special characters are used to aid in parsing the file and for error checking.

Timer

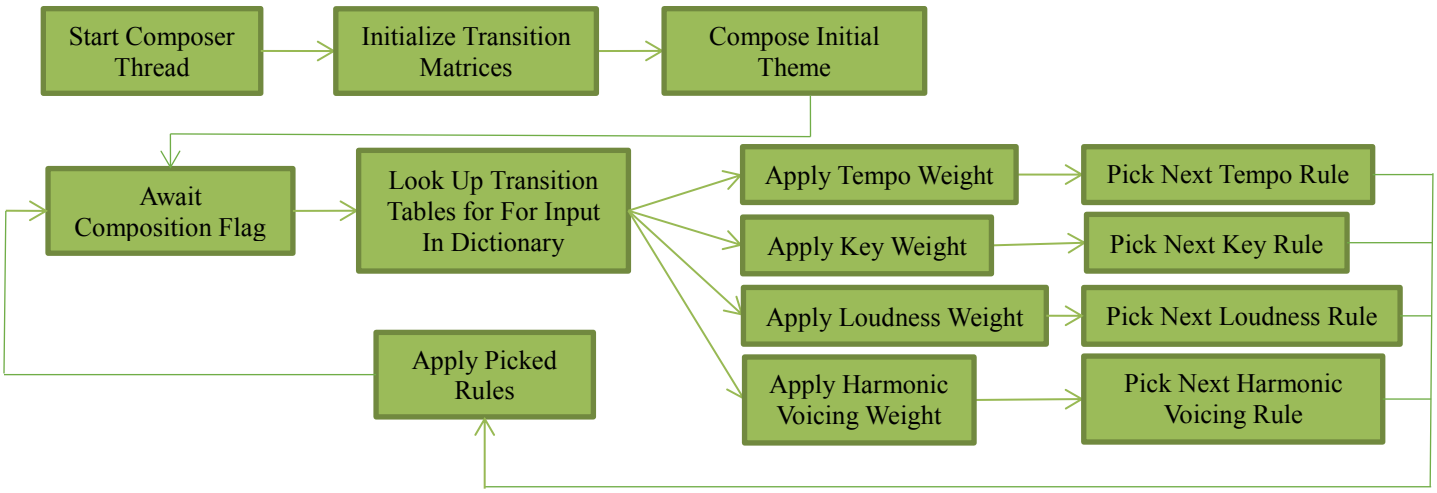
The timer task does not require a flow chart because it is only one job. A function, `timer_poll`, is registered with the system clock. On every 32nd note, `timer_poll` increments the current subdivision that the system is playing, and also logs the tick. The timer takes into account current tempo. The timer is relegated to a separate thread to enforce accuracy (if a player thread snags, the timer can still keep valid time and allow for a player thread to attempt to recover) as well as roughly mirror how music is often performed—tempo is often a discrete quality that is untied to the music presented on the page.

Input



The input section of the program handles user input, as comma separated list of styles that are currently saved in `style_hashtab` as names in `style_param_entry`. As the list of names needs to be accessible to more than one thread (written by input, read by composer), as well as the fact that it is of variable size based on input, a structure is used to capture it. This structure contains a `char**` along with the number of strings parsed from input. A global pointer references this structure. After the list is read and saved, the input thread toggles the `compose_flag`, which tells the composer that input has been entered and saved, so it should start working on a new theme.

Composer



The composer section of the software system composes each new theme that the player section plays, in response to the style parameter strings typed in by the user for input. After the `compose_flag` is released, the composer attempts to read the list of strings saved in the input thread structure, unless this is the first run. The first time the theme is heard is actually produced by randomly picking rules, before the user has

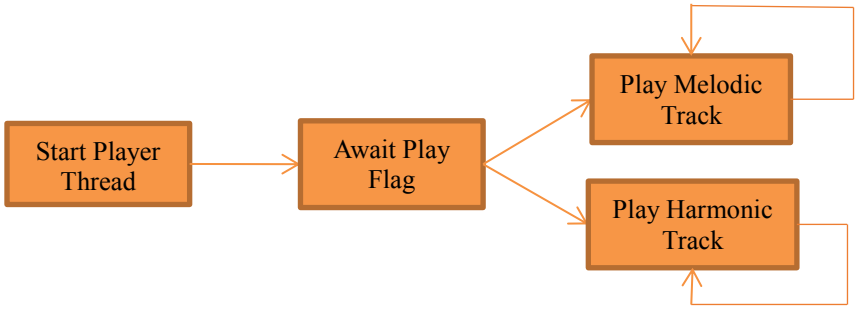
any chance to type input. Under this first run scenario, the composer thread initializes the theme's weight tables and then proceeds to the rule picking step.

For the next runs, however, the weight tables for each provided style string are looked up, and then multiplied into the applicable current weight table weight table. All of these sets of multiplication are independent, so although each style is calculated in sequence, each individual weight table can be modified in parallel, which speeds up the process. After each style parameter has been processed, then the values in the weight table are normalized and expressed as a range. These values are stored in the transition table for that expression parameter.

At this point, for every expression parameter that will be used in composing the next theme, a random number is generated, and then next composition rule for each expression parameter is chosen based on the previous composition rule. Just as before, because each expression parameter is independent of every other one, the selection can happen in parallel as well, along with random number generation.

At this point, the rules are applied to the theme, and a new variation is generated. It may seem like each rule can also be applied in parallel, but that, sadly, is not true. Several musical concepts have relationships that force the application of some rules to be in sequence. There is an intrinsic relationship between Harmonic Voicing and Key, for example. The key of the new variation must be chosen before the new Harmonic Voicing can be chosen. To avoid unneeded complexity, all rules were applied in sequence.

Player



The player part of the program plays the music written in the composer section. The composer writes to the track sections of the `MusicalTheme` data structure. The rules are applied directly to the tracks, they are modified as the player plays them. Due to the relative speed of track modification (at least for small themes) this generally does not lead to very much 'noise' or 'hiccups' in the sound. In fact, what is most often heard is a very fast blend of rules from several sources being played at once, which functions like an unstructured transition section into the new variation.

The playable tracks are arrays of note arrays, which themselves are just a set of pitches and a duration to play the pitches. This leads to several gross simplifications of what the player can play. Musical concepts that rely on tones changing at different times, such as suspensions, can't be played on one track. Several tracks could be used to allow for more complicated rhythmic structures. Each track, therefore, can be considered less as a line of sheet music and more as a set of continuous note events, where all events must change together.

Actually playing the notes in question is a basic task- the pitches saved are already mapped to the MIDI standard [12], saved as an integer between 0-127, rests

are saved as -1. Duration is expressed in terms of 32nd notes. Loudness is an integer, also in the 0-127 range, which determines the velocity on each note played for that theme. MIDI allows for another byte of expression information that is unused. The `Pm_WriteShort ()` function of the portmidi library is used to actually send the information to the sound card.

As each track plays independently, each track can be assigned a unique thread that handles the reading of notes from the track file, then the actual playing of those notes by interpreting them as midi data.

Conclusion

As this paper has shown, computers can react to high level style-based input in real time and compose a new variation appropriately. That being said, what this paper does not describe is a comprehensive system for artificial composition. All the values for the weight tables used for each style were human written, reflecting the author's own taste and approach to style. For purely algorithmic composition, the program would need some way to come up with the weight tables on its own. The leading concept is to create a fitness function that would allow a program to listen to music and categorize it according to a certain number of styles, then analyze each piece to figure out how the music of that category is being expressed, then build the weight tables from there.

But, even as a variation generator, the software system could be used to capture more parts of musical expression by adding more weight tables to each style and the required functions to compute how a change of state is reflected. First and foremost, ways to change and transform the melody line beyond a simple pitch translation would allow for each variation to sound very different. It generally follows that the more

parameters allowed to change from variation to variation will make each variation sound more musical.

This sort of automatic algorithmic musical composition can be used by composers as a composing aid, able to hear some samples of several different ways to write and hear the same theme. The program could also be used in entertainment, able to dynamically shift the soundtrack to keep one underlying theme as the tone of the theme changes according to whatever might be happening on screen. On a more abstract level, the approach this paper takes to abstract input can be used to allow for computer applications to try and understand higher level concepts as input, which could be used, for example, to alter the look and feel of an application based on how a user might describe their emotional state.

To sum it all up, I would say that I have learned a lot going through this project. From learning about Markov chains, to learning about parallel processing and multithreading, this project has probably been one of the hardest things I've embarked on to date. I'm thankful I got the chance to try and marry my love of computer science and my passion for music. I think that such opportunities are incredibly important, as C.P. Snow writes in "The Two Cultures", chances to bridge the 'academic gap' between the Sciences and the Arts are rare [11]. There is a gap between the sciences and the arts, and that gap shouldn't be there. Bridging that gap is important, and I am thankful I got such an opportunity.

References

[1] ISO/IEC 9899:TEC2, *Programming languages — C.*

[2] Kernighan, B., & Ritchie, D. (1988). *The C programming language* (2nd ed.). Upper Saddle River, NJ: Prentice Hall.

[3] PortMedia APIs. (n.d.). Retrieved March 17, 2013, from PortMedia website: http://portmedia.sourceforge.net/

[4] Norris, J. R. (1999). *Cambridge Series in Statistical and Probabilistic Mathematics: Vol. 2. Markov chains*. New York, NY: Cambridge University Press.

[6] Kostka, S., & Payne, D. (2009). *Tonal harmony* (Sixth ed.). New York, NY: McGraw-Hill.

[7] Cope, D. (2005). *Computer models of musical creativity.*

[8] Tempo. (n.d.). Retrieved March 17, 2013, from MuseScore website: http://musescore.org/en/handbook/tempo

[9] Kamien, Roger. *Music Appreciation* | *Brief*. 7th ed. New York: McGraw Hill, 2011. Print.

[10] Jacob, B. (1996). Algorithmic composition as a model of creativity.

[11] Snow, C. P. (1960). *The two cultures and the scientific revolution: the Rede Lecture 1959*. University Press.

[12] The MIDI Specification. (n.d.). Retrieved March 17, 2013, from http://www.bitter.com/~russtopia/MIDI/~jglatt/tech/mdidspec.htm

[13] Lejaren Hiller and Leonard Isaacson. Musical composition with a high-speed digital computer. *Journal of the Audio Engineering Society.* 1958.

[14] Robert Gjerdingen. Concrete musical knowledge and a computer program gor species counterpoint. In *Explorations in Music, the Arts, and Ideas: Essays in Honor of Leonard B Meyer.* Pendragon Press, Stuyvesant, 1988.

[15] Bell, C. (2011). Algorithmic music composition using dynamic Markov chains and genetic algorithms. *Journal of Computing Sciences in Colleges*, *27*(2), 99-107.

[16] Papadopoulos, G., & Wiggins, G. (1999). AI methods for algorithmic composition: A survey, a critical view and future prospects. In *AISB Symposium on Musical Creativity* (pp. 110-117). Edinburgh, UK.

APPROVAL SHEET

This is to certify that Johnathan Pagnutti has successfully completed his Senior Honors Thesis, entitled:

Real Time Algorithmic Musical Variation Using Style

Christopher M. Taylor — Directors of Thesis

Mahdi Abdelguerfi — Directors of Thesis

Charles L. Taylor — Second Reader

Abu Kabir Mostofa Sarwar — for the University Honors Program

April 19, 2013
Date